Network Working Group
Request for Comments: 963

Deepinder P. Sidhu
Iowa State University
November 1985

SOME PROBLEMS WITH THE SPECIFICATION OF THE MILITARY STANDARD INTERNET PROTOCOL

STATUS OF THIS MEMO

The purpose of this RFC is to provide helpful information on the Military Standard Internet Protocol (MIL-STD-1777) so that one can obtain a reliable implementation of this protocol standard. Distribution of this note is unlimited.

ABSTRACT

This paper points out several significant problems in the specification of the Military Standard Internet Protocol (MIL-STD-1777, dated August 1983 [MILS83a]). These results are based on an initial investigation of this protocol standard. The problems are: (1) a failure to reassemble fragmented messages completely; (2) a missing state transition; (3) errors in testing for reassembly completion; (4) errors in computing fragment sizes; (5) minor errors in message reassembly; (6) incorrectly computed length for certain datagrams. This note also proposes solutions to these problems.

1. Introduction

In recent years, much progress has been made in creating an integrated set of tools for developing reliable communication protocols. These tools provide assistance in the specification, verification, implementation and testing of protocols. Several protocols have been analyzed and developed using such tools. Examples of automated verification and implementation of several real world protocols are discussed in [BLUT82] [BLUT83] [SIDD83] [SIDD84].

We are currently working on the automatic implementation of the Military Standard Internet Protocol (IP). This analysis will be based on the published specification [MILS83a] of IP dated 12 August 1983.

While studying the MIL Standard IP specification, we have noticed numerous errors in the specification of this protocol. One consequence of these errors is that the protocol will never deliver fragmented incoming datagrams; if this error is corrected, such datagrams will be missing some data and their lengths will be incorrectly reported. In addition, outgoing datagrams that are divided into fragments will be missing some data. The proof of these statements follows from the specification of IP [MILS83a] as discussed below.

2. Internet Protocol

The Internet Protocol (IP) is a network layer protocol in the DoD protocol hierarchy which provides communication across interconnected packet-switched networks in an internetwork environment. IP provides a pure datagram service with no mechanism for reliability, flow control, sequencing, etc. Instead, these features are provided by a connection-oriented protocol, DoD Transmission Control Protocol (TCP) [MILS83b], which is implemented in the layer above IP. TCP is designed to operate successfully over channels that are inherently unreliable, i.e., which can lose, damage, duplicate, and reorder packets.

Over the years, DARPA has supported specifications of several versions of IP; the last one appeared in [POSJ81]. A few years ago, the Defense Communications Agency decided to standardize IP for use in DoD networks. For this purpose, the DCA supported formal specification of this protocol, following the design discussed in [POSJ81] and the technique and organization defined in [SDC82]. A detailed specification of this protocol, given in [MILS83a], has been adopted as the DoD standard for the Internet Protocol.

The specification of IP state transitions is organized into decision tables; the decision functions and action procedures are specified in a subset of Ada[1], and may employ a set of machine-specific data structures. Decision tables are supplied for the pairs <state name, interface event> as follows: <inactive, send from upper layer>, <inactive, receive from lower layer>, and <reassembling, receive from lower layer>. To provide an error indication in the case that some fragments of a datagram are received but some are missing, a decision table is also supplied for the pair <reassembling, reassembly time limit elapsed>. (The event names are English descriptions and not the names employed by [MILS83a].)

3. Problems with MIL Standard IP

One of the major functions of IP is the fragmentation of datagrams that cannot be transmitted over a subnetwork in one piece, and their subsequent reassembly. The specification has several problems in this area. One of the most significant is the failure to insert the last fragment of an incoming datagram; this would cause datagrams to be delivered to the upper-level protocol (ULP) with some data missing. Another error in this area is that an incorrect value of the data length for reassembled datagrams is passed to the ULP, with unpredictable consequences.

As the specification [MILS83a] is now written, these errors are of

little consequence, since the test for reassembly completion will always fail, with the result that reassembled datagrams would never be delivered at all.

In addition, a missing row in one of the decision tables creates the problem that network control (ICMP) messages that arrive in fragments will never be processed. Among the other errors are the possibility that a few bytes will be discarded from each fragment transmitted and certain statements that will create run-time exceptions instead of performing their intended functions.

A general problem with this specification is that the program language and action table portions of the specification were clearly not checked by any automatic syntax checking process. Variable and procedure names are occasionally misspelled, and the syntax of the action statements is often incorrect. We have enumerated some of these problems below as a set of cautionary notes to implementors, but we do not claim to have listed them all. In particular, syntax errors are only discussed when they occur in conjunction with other problems.

The following section discusses some of the serious errors that we have discovered with the MIL standard IP [MIL83a] during our initial study of this protocol. We also propose corrections to each of these problems.

4. Detailed Discussion of the Problems

Problem 1: Failure to Insert Last Fragment

This problem occurs in the decision table corresponding to the state reassembling and the input "receive from lower layer" [MILS83a, sec 9.4.6.1.3]. The problem occurs in the following row of this table:[2]

check-sum valid?	SNP params valid?	TTL valid ?	where to ?	a frag ?	reass done ?	ICMP check-sum?	
YES	YES	YES	ULP	YES	YES	d	reass_delivery; state := INACTIVE

The reass_done function, as will be seen below, returns YES if the

fragment just received is the last fragment needed to assemble a complete datagram and NO otherwise. The action procedure reass_delivery simply delivers a completely reassembled datagram to the upper-level protocol. It is the action procedure reassemble that inserts an incoming fragment into the datagram being assembled. Since this row does not call reassemble, the result will be that every incoming fragmented datagram will be delivered to the upper layer with one fragment missing. The solution is to rewrite this row of the table as follows:

check-sum valid?	SNP params valid?	TTL valid ?	where to ?	a frag ?	reass done ?	ICMP check-sum?	
YES	YES	YES	ULP	YES	YES	d	reassemble; reass_ delivery; state := INACTIVE

Incidentally, the mnemonic value of the name of the reass_done function is questionable, since at the moment this function is called datagram reassembly cannot possibly have been completed. A better name for this function might be last_fragment.

Problem 2: Missing State Transition

This problem is the omission of a row of the same decision table [MILS83a, sec 9.4.6.1.3]. Incoming packets may be directed to an upper-level protocol (ULP), or they may be network control messages, which are marked ICMP (Internet Control Message Protocol). When control messages have been completely assembled, they are processed by an IP procedure called analyze. The decision table contains the row

check-sum valid?	SNP params valid?	TTL valid ?	where to ?	a frag ?	reass done ?	ICMP check-sum?	
YES	YES	YES	ICMP	YES	NO	d	reassemble;

but makes no provision for the case in which where_to returns ICMP, a_frag returns YES, and reass_done returns YES. An additional row should be inserted, which reads as follows:

check-sum valid?	SNP params valid?	TTL valid ?	where to ?	a frag ?	reass done ?	ICMP check-sum?	
YES	YES	YES	ICMP	YES	YES	d	reassemble; analyze; state := INACTIVE

Omitting this row means that incoming fragmented ICMP messages will never be analyzed, since the state machine does not have any action specified when the last fragment is received.

Problem 3: Errors in reass_done

The function reass_done, as can be seen from the above, determines whether the incoming subnetwork packet contains the last fragment needed to complete the reassembly of an IP datagram. In order to understand the errors in this function, we must first understand how it employs its data structures.

The reassembly of incoming fragments is accomplished by means of a bit map maintained separately for each state machine. Since all fragments are not necessarily the same length, each bit in the map represents not a fragment, but a block, that is, a unit of eight octets. Each fragment, with the possible exception of the "tail" fragment (we shall define this term below), is an integral number of consecutive blocks. Each fragment's offset from the beginning of the datagram is given, in units of blocks, by a field in the packet header of each incoming packet. The total length of each fragment, including the fragment's header, is specified in the header field total_length; this length is given in octets. The length of the header is specified in the field header_length; this length is given in words, that is, units of four octets.

In analyzing this subroutine, we must distinguish between the "tail" fragment and the "last" fragment. We define the last fragment as the one which is received last in time, that is, the fragment that permits reassembly to be completed. The tail fragment is the fragment that is spatially last, that is, the fragment that is spatially located after any other fragment. The

length and offset of the tail fragment make it possible to compute the length of the entire datagram. This computation is actually done in the action procedure reassembly, and the result is saved in the state vector field total_data_length; if the tail fragment has not been received, this value is assumed to be zero.

It is the task of the reass_done function [MILS83a, sec 9.4.6.2.6] to determine whether the incoming fragment is the last fragment. This determination is made as follows:

1) If the tail fragment has not been received previously and the incoming fragment is not the tail fragment, then return NO.

2) Otherwise, if the tail fragment has not been received, but the incoming fragment is the tail fragment, determine whether all fragments spatially preceding the tail fragment have also been received.

3) Otherwise, if the tail fragment has been received earlier, determine whether the incoming fragment is the last one needed to complete reassembly.

The evaluation of case (2) is accomplished by the following statment:

```
if (state_vector.reassembly_map from 0 to
  (((from_SNP.dtgm.total_length -
      (from_SNP.dtgm.header_length * 4) + 7) / 8)
  + 7) / 8 is set)
then return YES;
```

The double occurrence of the subexpression " + 7) / 8" is apparently a misprint. The function f(x) = (x + 7) / 8 will convert x from octets to blocks, rounding any remainder upward. There is no need for this function to be performed twice. The second problem is that the fragment_offset field of the incoming packet is ignored. The tail fragment specifies only its own length, not the length of the entire datagram; to determine the latter, the tail fragment's offset must be added to the tail fragment's own length. The third problem hinges on the meaning of the English "... from ... to ..." phrase. If this phrase has the same meaning as the ".." range indication in Ada [ADA83, sec 3.6], that is, includes both the upper and lower bounds, then it is necessary to subtract 1 from the final expression.

The expression following the word to, above, should thus be changed to read

```
   from_SNP.dtgm.fragment_offset +
      ((from_SNP.dtgm.total_length -
         (from_SNP.dtgm.header_length * 4) + 7) / 8) - 1
```

Another serious problem with this routine occurs when evaluating case (3). In this case, the relevant statement is

```
   if (all reassembly map from 0 to
     (state_vector.total_data_length + 7)/8 is set
   then return YES
```

If the tail fragment was received earlier, the code asks, in effect, whether all the bits in the reassembly map have been set. This, however, will not be the case even if the incoming fragment is the last fragment, since the routine reassembly, which actually sets these bits, has not yet been called for this fragment. This statement must therefore skip the bits corresponding to the incoming fragment. In specifying the range to be tested, allowance must be made for whether these bits fall at the beginning of the bit map or in the middle (the case where they fall at the end has already been tested). The statement must therefore be changed to read

```
   if from_SNP.dtgm.fragment_offset = 0 then
     if (all reassembly map from
       from_SNP.dtgm.fragment_offset +
       ((from_SNP.dtgm.total_length -
         from_SNP.dtgm.header_length * 4) + 7) / 8
     to ((state_vector.total_data_length + 7) / 8 - 1) is set)
   then return YES;
   else return NO;
   end if;

   else
   if (all reassembly map from 0 to
     (from_SNP.dtgm.fragment_offset - 1) is set)
     and (all reassembly map from
       from_SNP.dtgm.fragment_offset +
         ((from_SNP.dtgm.total_length -
           from_SNP.dtgm.header_length * 4) + 7) / 8
       to ((state_vector.total_data_length + 7) / 8 - 1) is set)
   then return YES;
   else return NO;
   end if;
   end if;
```

Note that here again it is necessary to subtract 1 from the upper bound.

Problem 4: Errors in fragment_and_send

The action procedure fragment_and_send [MILS83a, sec 9.4.6.3.7] is used to break up datagrams that are too large to be sent through the subnetwork as a single packet. The specification requires [MILS83a sec 9.2.2, sec 9.4.6.3.7] each fragment, except possibly the "tail" fragment, to contain a whole number of 8-octet groups (called "blocks"); moreover, each fragment must begin at a block boundary.

In the algorithm set forth in fragment_and_send, all fragments except the tail fragment are set to the same size; the procedure begins by calculating this size. This is done by the following statement:

```
data_per_fragment := maximum subnet transmission unit
                        - (20 + number of bytes of option data);
```

Besides the failure to allow for header padding, which is discussed in the next section, this statement makes the serious error of not assuring that the result is an integral multiple of the block size, i.e., a multiple of eight octets. The consequence of this would be that as many as seven octets per fragment would never be sent at all. To correct this problem, and to allow for header padding, this statement must be changed to

```
data_per_fragment := (maximum subnet transmission unit
        - (((20 + number of bytes of option data)+3)/4*4)/8*8;
```

Another problem in this procedure is the failure to provide for the case in which the length of the data is an exact multiple of eight. The procedure contains the statements

```
number_of fragments := (from_ULP.length +
                (data_per_fragment - 1)) / data_per_fragment;

data_in_last_frag := from_ULP.length modulo data_per_fragment;
```

(Note that in our terminology we would rename data_in_last_frag as data_in_tail_frag; notice, also, that the proper spelling of the Ada operator is mod [ADA83, sec 4.5.5].)

If data_in_last_frag is zero, some serious difficulties arise. One result might be that the datagram will be broken into one more

fragment than necessary, with the tail fragment containing no data bytes. The assignment of data into the tail fragment will succeed even though it will now take the form

```
output_data [i..i-1] := input_data [j..j-1];
```

because Ada makes provision for so-called "null slices" [ADA83, sec 4.1.2] and will treat this assignment as a no-op [ADA83, sec 5.2.1].

This does, however, cause the transmission of an unnecessary packet, and also creates difficulties for the reassembly procedure, which must now be prepared to handle empty packets, for which not even one bit of the reassembly map should be set. Moreover, as the procedure is now written, even this will not occur. This is because the calculation of the number of fragments is incorrect.

A numerical example will clarify this point. Suppose that the total datagram length is 16 bytes and that the number of bytes per fragment is to be 8. Then the above statements will compute number_of_fragments = (16 + 7)/8 = 2 and data_in_last_frag = 16 mod 8 = 0. The result of the inconsistency between number_of_fragments and data_in_last_frag will be that instead of sending three fragments, of lengths 8, 8, and 0, the procedure will send only two fragments, of lengths 8 and 0; the last eight octets will never be sent.

To avoid these difficulties, the specification should add the following statement, immediately after computing data_in_last_frag:

```
if data_in_last_frag = 0 then
                        data_in_last_frag := data_per_fragment;
end if;
```

This procedure also contains several minor errors. In addition to failures to account for packet header padding, which are enumerated in the next section, there is a failure to convert the header length from words (four octets) to octets in one statement. This statement, which calculates the total length of the non-tail fragments, is

```
to_SNP.dtgm.total_length := to_SNP.dtgm.header_length
                                        + data_per_fragment;
```

Since header length is expressed in units of words, this statement should read

```
to_SNP.dtgm.total_length := to_SNP.dtgm.header_length * 4
                                        + data_per_fragment;
```

This is apparently no more than a misprint, since the corresponding calculation for the tail fragment is done correctly.

Problem 5: Errors in reassembly

The action procedure reassembly [MILS83a, sec 9.4.6.3.9], which is referred to as reassemble elsewhere in the specification [MILS83a, sec 9.4.6.1.2, sec 9.4.6.1.3], inserts an incoming fragment into a datagram being reassembled. This procedure contains several relatively minor errors.

In two places in this procedure, a range is written to contain one more member than it ought to have. In the first, data from the fragment is to be inserted into the datagram being reassembled:

```
state_vector.data [from_SNP.dtgm.fragment_offset*8 ..
    from_SNP.dtgm.fragment_offset*8 + data_in_frag] :=
            from_SNP.dtgm.data [0..data_in_frag-1];
```

In this statement, the slice on the left contains one more byte than the slice on the right. This will cause a run-time exception to be raised [ADA83, sec 5.2.1]. The statement should read

```
state_vector.data [from_SNP.dtgm.fragment_offset*8 ..
    from_SNP.dtgm.fragment_offset*8 + data_in_frag - 1] :=
            from_SNP.dtgm.data [0..data_in_frag-1];
```

A similar problem occurs in the computation of the range of bits in the reassembly map that corresponds to the incoming fragment. This statement begins

```
for j in (from_SNP.dtgm.fragment_offset) ..
         ((from_SNP.dtgm.fragment_offset +
       data_in_frag + 7)/8) loop
```

Not only are the parentheses in this statement located incorrectly (because the function f(x) = (x + 7) / 8 should be executed only on the argument data_in_frag), but also this range contains one extra member. The statement should read

```
for j in (from_SNP.dtgm.fragment_offset) ..
         (from_SNP.dtgm.fragment_offset +
        (data_in_frag + 7)/8) - 1 loop
```

Note that if the statement is corrected in this manner it will also handle the case of a zero-length fragment, mentioned above, since the loop will not be executed even once [ADA83, sS 5.5].

Another minor problem occurs when this procedure attempts to save the header of the leading fragment. The relevant statement is

```
state_vector.header := from_SNP.dtgm;
```

This statement attempts to transfer the entire incoming fragment into a record that is big enough to contain only the header. The result, in Ada, is not truncation, but a run-time exception [ADA83, sec 5.2]. The correction should be something like

```
state_vector.header := from_SNP.dtgm.header;
```

This correction cannot be made without also defining the header portion of the datagram as a subrecord in [MILS83a, sec. 9.4.4.6]; such a definition would also necessitate changing many other statements. For example, from_SNP.dtgm.fragment_offset would now have to be written as from_SNP.dtgm.header.fragment_offset.

Another possible solution is to write the above statement as a series of assignments for each field in the header, in the following fashion:

```
state_vector.header.version :=
                                    from_SNP.dtgm.version;
state_vector.header.header_length :=
                                  from_SNP.dtgm.header_length;
state_vector.header.type_of_service :=
                                from_SNP.dtgm.type_of_service;

-- etc.
```

Note also that this procedure will fail if an incoming fragment, other than the tail fragment, does not contain a multiple of eight characters. Implementors must be careful to check for this in the decision function SNP_params_valid [MILS83a, sec 9.4.6.2.7].

Problem 6: Incorrect Data Length for Fragmented Datagrams

The procedure reassembled_delivery [MILS83a, sec 9.4.6.3.10] does not deliver the proper data length to the upper-level protocol. This is because the assignment is

```
to_ULP.length := state_vector.header.total_length
                 - state_vector.header.header_length * 4;
```

The fields in state_vector.header have been filled in by the reassembly procedure, discussed above, by copying the header of the leading fragment. The field total_length in this fragment, however, refers only to this particular fragment, and not to the entire datagram (this is not entirely clear from it definition in [MILS83a, sec 9.3.4], but the fragment_and_send procedure [MILS83a, sec 9.4.6.3.7] insures that this is the case).

The length of the entire datagram can only be computed from the length and offset of the tail fragment. This computation is actually done in the reassembly procedure [MILS83a, sec 9.4.6.3.9], and the result saved in state_vector.total_data_length (see above). It is impossible, however, for reassembly to fill in state_vector.header.total_length at this time, because state_vector.header.header_length is filled in from the lead fragment, which may not yet have been received.

Therefore, reassembled_delivery must replace the above statement with

```
to_ULP.length := state_vector.total_data_length;
```

The consequence of leaving this error uncorrected is that the upper-level protocol will be informed only of the delivery of as many octets as there are in the lead fragment.

5. Implementation Difficulties of MIL Standard IP

In addition to the problems discussed above, there are several features of the MIL standard IP specification [MILS83a] which lead to difficulties for the implementor. These difficulties, while not actually errors in the specification, take the form of assumptions which are not explicitly stated, but of which implementors must be aware.

5.1 Header Padding

In several places, the specification makes a computation of the length of a packet header without explicitly allowing for padding. The padding is needed because the specification requires [MILS83a, sec 9.3.14] that each header end on a 32-bit boundary.

One place this problem arises is in the need_to_frag decision function [MILS83a, sec 9.4.6.2.5]. This function is used to determine whether fragmentation is required for an outgoing datagram. It consists of the single statement

```
if ((from_ULP.length + (number of bytes of option data)
      + 20) > maximum transmission unit of the local subnetwork
then return YES
else return NO;
end if;
```

(A minor syntax error results from not terminating the first return statement with a semicolon [ADA83, sec 5.1, sec 5.3, sec 5.9].) In order to allow for padding, the expression for the length of the outgoing datagram should be

```
(((from_ULP.length + (number of bytes of option data) + 20)
                                                  + 3)/4 * 4)
```

Another place that this problem arises is in the action procedure build_and_send [MILS83a, sec 9.4.6.3.2], which prepares unfragmented datagrams for transmission. To compute the header field header_length, which is expressed in words, i.e., units of four octets [MILS83a, sec 9.3.2], this procedure contains the statement

```
to_SNP.dtgm.header_length := 5 +
                              (number of bytes of option data)/4;
```

In order to allow for padding, this statement should read

```
to_SNP.dtgm.header_length :=
                        5 + ((number of bytes of option data)+3)/4;
```

The identical statement appears in the action procedure fragment_and_send [MILS83a, sec 9.4.6.3.7], which prepares datagram fragments for transmission, and requires the same correction.

The procedure fragment_and_send also has this problem in two other places. In the first, the number of octets in each fragment is computed by

```
data_per_fragment := maximum subnet transmission unit
                        - (20 + number of bytes of option data);
```

In order to allow for padding, this statement should read

```
data_per_fragment := maximum subnet transmission unit
                - (((20 + number of bytes of option data)+3)/4*4);
```

(Actually, this statement must be changed to

```
data_per_fragment := (maximum subnet transmission unit
                - (((20 + number of bytes of option data)+3)/4*4)/8*8;
```

in order to accomplish its intended purpose, for reasons which have been discussed above.)

A similar problem occurs in the statement which computes the header length for individual fragments:

```
to_SNP.dtgm.header_length := 5 +
                        (number of copy options octets/4);
```

To allow for padding, this should be changed to

```
to_SNP.dtgm.header_length := 5 +
                        (number of copy options octets+3/4);
```

Notice that all of these errors can also be corrected if the English phrase "number of bytes of option data", and similar phrases, are always understood to include any necessary padding.

5.2 Subnetworks with Small Transmission Sizes

When an outgoing datagram is too large to be transmitted as a single packet, it must be fragmented. On certain subnetworks, the possibility exists that the maximum number of bytes that may be transmitted at a time is less than the size of an IP packet header for a given datagram. In this case, the datagram cannot be sent, even in fragmented form. Note that this does not necessarily mean that the subnetwork cannot send any datagrams at all, since the size of the header may be highly variable. When this problem arises, it should be detected by IP. The proper place to detect this situation is in the function can_frag.

The can_frag decision function [MILS83a, sec 9.4.6.2.2] is used to determine whether a particular outgoing datagram, which is too long to be transmitted as a single fragment, is allowed to be fragmented. In the current specification, this function consists of the single statement

```
if (from_ULP.dont_fragment = TRUE)
then return NO
else return YES
end if;
```

(A minor syntax error is that the return statements should be terminated by semicolons; see [ADA83, sec 5.1, sec 5.3, sec 5.9].)

If the above problem occurs, the procedure fragment_and_send will obtain negative numbers for fragment sizes, with unpredictable results. This should be prevented by assuring that the subnetwork can send the datagram header and at least one block (eight octets) of data. The can_frag function should be recoded as

```
if ((8 + ((number of bytes of option data)+3)/4*4 + 20)
          > maximum transmission unit of the local subnetwork)
then return NO;
elsif (from_ULP.dont_fragment = TRUE)
then return NO
else return YES
end if;
```

This is similar to the logic of the function need_to_frag, discussed above.

5.3 Subnetwork Interface

Provision is made for the subnetwork to report errors to IP [MILS83a, sec 6.3.6.2], but no provision is made for the IP entity to take any action when such errors occur.

In addition, the specification [MILS83a, sec 8.2.1.1] calls for the subnetwork to accept type-of-service indicators (precedence, reliability, delay, and throughput), which may be difficult to implement on many local networks.

5.4 ULP Errors

The IP specification [MILS83a, sec 9.4.6.3.6] states

> The format of error reports to a ULP is implementation dependent. However, included in the report should be a value indicating the type of error, and some information to identify the associated data or datagram.

The most natural way to provide the latter information would be to return the datagram identifier to the upper-level protocol, since this identifier is normally supplied by the sending ULP [MILS83a, sec 9.3.5]. However, the to_ULP data structure makes no provision for this information [MILS83a, sec 9.4.4.3], probably because this information is irrelevant for datagrams received from the subnetwork. Implementors may feel a need to add this field to the to_ULP data structure.

5.5 Initialization of Data Structures

The decision function reass_done [MILS83a, sec 9.4.6.2.6] makes the implicit assumption that data structures within each finite state machine are initialized to zero when the machine is created. In particular, this routine will not function properly unless state_vector.reassembly_map and state_vector.total_data_length are so initialized. Since this assumption is not stated explicitly, implementors should be aware of it. There may be other initialization assumptions that we have not discovered.

5.6 Locally Defined Types

The procedures error_to_source [MILS83a, sec 9.4.6.3.5] and error_to_ULP [MILS83a, sec 9.4.6.3.6] define enumeration types in comments. The former contains the comment

```
error_param : (PARAM_PROBLEM, EXPIRED_TTL, PROTOCOL_UNREACH);
```

and the latter

```
error_param : (PARAM_PROBLEM, CAN'T_FRAGMENT, NET_UNREACH,
                              PROTOCOL_UNREACH, PORT_UNREACH);
```

These enumerated values are used before they are encountered [MILS83a, sec 9.4.6.1.1, sec 9.4.6.1.2, sec 9.4.6.1.3, et al.]; implementors will probably wish to define some error type globally.

5.7 Miscellaneous Difficulties

The specification contains many Ada syntax errors, some of which have been shown above. We have only mentioned syntax errors above, however, when they occurred in conjunction with other problems. One of the main syntactic difficulties that we have not mentioned is that the specification frequently creates unnamed types, by declaring records within records; such declarations are legal in Pascal, but not in Ada [ADA83, sec 3.7].

Another problem is that slice assignments frequently do not contain the same number of elements on the left and right sides, which will raise a run-time exception [ADA83, sec 5.2.1]. While we have mentioned some of these, there are others which are not enumerated above.

In particular, the procedure error_to_source [MILS83a, sec 9.4.6.3.5] contains the statement

```
to_SNP.dtgm.data [8..N+3] := from_SNP.dtgm.data [0..N-1];
```

We believe that N+3 is a misprint for N+8, but even so the left side contains one more byte than the right. Implementors should carefully check every slice assignment.

6. An Implementation of MIL Standard IP

In our discussion above, we have pointed out several serious problems with the Military Standard IP [MILS83a] specification which must be corrected to produce a running implementation conforming to this standard. We have produced a running C implementation for the MIL Standard IP, after problems discussed above were fixed in the IP specification. An important feature of this implementation is that it was generated semi-automatically from the IP specification with the help of a protocol development system [BLUT82] [BLUT83] [SIDD83]. Since this implementation was derived directly from the IP specification with the help of tools, it conforms to the IP standard better that any handed-coded IP implementation can do.

The problems pointed out in this paper with the current specification of the MIL Standard IP [MILS83a] are based on an initial investigation of the protocol.

NOTES

[1] Ada is a registered trademark of the U.S. Government - Ada Joint Program Office.

[2] d indicates a "don't care" condition.

ACKNOWLEDGEMENTS

The author extends his gratitude to Tom Blumer Michael Breslin, Bob Pollack and Mark J. Vincenzes, for many helpful discussions. Thanks are also due to B. Simon and M. Bernstein for bringing to author's attention a specification of the DoD Internet Protocol during 1981-82 when a detailed study of this protocol began. The author is also grateful to Jon Postel and Carl Sunshine for several informative discussions about DoD IP/TCP during the last few years.

REFERENCES

[ADA83] Military Standard Ada(R) Programming Language, United States Department of Defense, ANSI/MIL-STD-1815A-1983, 22 January 1983

[BLUT83] Blumer, T. P., and Sidhu, D. P., "Mechanical Verification and Automatic Implementation of Communication Protocols," to appear in IEEE Trans. Softw. Eng.

[BLUT82] Blumer, T. P., and Tenney, R. L., "A Formal Specification Technique and Implementation Method for Protocols," Computer Networks, Vol. 6, No. 3, July 1982, pp. 201-217.

[MILS83a] "Military Standard Internet Protocol," United States Department of Defense, MIL-STD-1777, 12 August 1983.

[MILS83b] "Military Standard Transmission Control Protocol," United States Department of Defense, MIL-STD-1778, 12 August 1983.

[POSJ81] Postel, J. (ed.), "DoD Standard Internet Protocol," Defense Advanced Research Projects Agency, Information Processing Techniques Office, RFC-791, September 1981.

[SDC82] DCEC Protocol Standardization Program: Protocol Specification Report, System Development Corporation, TM-7172/301/00, 29 March 1982

[SIDD83] Sidhu, D. P., and Blumer, T. P., "Verification of NBS Class 4 Transport Protocol," to appear in IEEE Trans. Comm.

[SIDD84] Sidhu, D. P., and Blumer, T. P., "Some Problems with the Specification of the Military Standard Transmission Control Protocol," in Protocol Specification, Testing and Verification IV, (ed.) Y. Yemini et al (1984).